

# **DEEP LEARNING METHOD FOR DENOISING MONTE CARLO RENDERS FOR VR APPLICATIONS**

An Undergraduate Research Scholars Thesis

by

AKSEL TAYLAN

Submitted to the Undergraduate Research Scholars program at  
Texas A&M University  
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor:

Dr. Nima Kalantari

May 2020

Major: Computer Science

# TABLE OF CONTENTS

	Page
ABSTRACT.....	1
DEDICATION.....	2
ACKNOWLEDGMENTS .....	3
NOMENCLATURE .....	4
CHAPTER	
I.    INTRODUCTION .....	5
History.....	5
Ray Tracing.....	6
Monte Carlo Method.....	8
Denoising .....	9
VR.....	10
II.   METHODS .....	11
Dataset.....	11
Network Architecture.....	14
Experiment 1 – Concatenated Features.....	15
Experiment 2 – Separate Encoders .....	16
Performance Evaluation.....	17
III.  RESULTS .....	19
Numerical Analysis.....	19
Visual Analysis .....	19
Validation Analysis.....	22
IV.  CONCLUSION & FUTURE WORK.....	23
Conclusion .....	23
Future Work .....	24
REFERENCES .....	26

# **ABSTRACT**

## **Deep Learning Method for Denoising Monte Carlo Renders for VR Applications**

Aksel Taylan  
Department of Computer Science & Engineering  
Texas A&M University

Research Advisor: Dr. Nima Kalantari  
Department of Computer Science & Engineering  
Texas A&M University

Monte Carlo path tracing is one of the most desirable methods to render an image from three-dimensional data thanks to its innate ability to portray physically realistic and desirable phenomena such as soft shadows, motion blur, and global illumination. Due to the nature of the algorithm, it is extremely computationally expensive to produce a converged image.

A commonly researched and proposed solution to the enormous time cost of rendering an image using Monte Carlo path tracing is denoising. This entails quickly rendering a noisy image with a low sample count and using a denoising algorithm to eradicate noise and deliver a clean image that is comparable to the ground truth. Many such algorithms focus on general image filtering techniques, and others lean on the power of deep learning.

This thesis explores methods utilizing deep learning to denoise Monte Carlo renders for virtual reality applications. Images for virtual reality, or ‘VR’ are composed of both a ‘left eye’ and ‘right eye’ image, doubling the computation cost of rendering and subsequent denoising. The methods tested in this thesis attempt to utilize stereoscopic image data to enhance denoising results from convolutional neural networks.

## **DEDICATION**

To my family and friends, who have supported me along my journey through academia.

## **ACKNOWLEDGMENTS**

I would like to thank my advisor, Dr. Nima Kalantari, for allowing me to partake research with him and guiding me through the process. I would also like to acknowledge my friend and fellow researcher Nicholas Milef, who was an instrumental part in the process of learning and progressing through this project.

Finally, many thanks to my mother, father, and brother for being an amazing family and lifting me up during tough times.

## NOMENCLATURE

GT	Ground truth
SPP	Samples per pixel
FOV	Field of view
RGB	Red Green Blue (color)
2D/3D	Two-dimensional/Three-dimensional
GPU/CPU	Graphics processing unit/Central processing unit
VR	Virtual reality

# **CHAPTER I**

## **INTRODUCTION**

Rendering is the process of transforming 3D data into a 2D image. In industries such as computer animation, video games, architectural visualization, and more, 3D data is created through the use of various software packages such as Maya, Houdini, or Cinema4D, and textures and lighting are applied to objects and scenes to define how these 3D models will look. Rendering utilizes all of this information and through various algorithms, computes what the color of every pixel in the final image should be.

### **History**

There have been several different types of rendering algorithms over time. Early on, the types of algorithms being developed were heavily restricted by the power of hardware at the time. GPUs and CPUs weren't where they are now, and so engineers and scientists had to think of different ways to produce imagery. The first majorly popular rendering algorithm was rasterization, a method still used to this day for various purposes. Rasterization involves projecting objects in a 3D scene to a 2D representation using perspective projection, and then determines if certain pixels are contained within the resulting projection. Due to its fast nature, this algorithm is the standard technique to render video games, as video games need to adhere to a strict frame rates (e.g., 60 frames-per-second) to maintain viewer coherency and stability. With other industries and products such as computer animation, this is not a problem, as there is no real-time aspect to a feature film. However, the constraints of hardware and memory management meant that rasterization or some modification of it was used for those fields as well.

In fact, researchers at this time were set on developing algorithms to best utilize the resources available to them, often sacrificing physical accuracy for speed. As is explored in the recent paper on the history of Pixar's famous RenderMan rendering software [1], the early REYES algorithm used by Pixar to produce their first feature film *Toy Story* involved optimizing memory usage and quick texture access, among other features geared towards producing images as fast as possible. Though the motion blur and depth of field effects were not accurate to the real-world, at this point in time, the performance tradeoff was necessary. There were, of course, drawbacks to such an approach. REYES had no way to naturally determine shadows, and thus computing shadow maps were necessary. Shadow maps required a host of tricks and cheats to pull off as the complexity of lights and interactions in the scene rose. A similar problem existed with reflections, which also had to be rendered with maps, as out of the box REYES could not handle this type of effect.

Path tracing was actually a well-known algorithm [6] at the time, but the computational complexity was too expensive for contemporary hardware. As hardware progressed, algorithms improved and more innovative memory models were discovered, path-tracing became more and more of a possibility. Over the years, ray tracing slowly gained steam, until eventually it was at the forefront of the industry.

## **Ray Tracing**

Ray tracing is an alternative rendering algorithm compared to rasterization, and it works in an almost completely opposite manner. To determine the color of every pixel in a final image, rays are shot from the camera into the 3D scene, and based on what objects the ray intersects, shading calculations are performed, and by shooting rays for every pixel, you can generate your image [6]. In the early days of computer graphics, this method of rendering was considered too



costly, as checking for intersections is a fairly expensive operation for the computer, and in order to render a scene with a large amount of objects, in its most primitive form, each ray would need to check if it had intersected with millions of different triangles that made up the objects in the scene. Although there was potential, ray tracing was mostly cast aside and only used on simple static scenes situationally. However, due to its ability to cast shadows and simulate reflections and refractions easily, ray tracing continued to generate buzz. One particular subset of ray tracing became particularly noteworthy, and would go on to sit at the forefront of rendering as we know it today.

### *Path Tracing*

Path tracing inherits the core methodology of ray tracing, but it truly shines in its novel features that closely resemble how light interacts with objects in the real-world [7]. In a path tracing algorithm, once a ray intersects an object, shading calculations are performed, as per any ray tracing algorithm, and then the ray stochastically bounces off that object and travels in another direction, until it intersects with something else. There, the ray performs more calculations, and bounces, and so on. In real life, light bounces an infinite amount of times, creating a phenomenon called indirect lighting or global illumination. Essentially, this refers to the fact that all objects in the real world are secondary light sources. Because we are so used to this effect, it may be hard to notice, but it's why the world isn't pitch black when you turn the lights off at night. Objects in the world around you are emitting small amounts of light that carry over to other objects around them, giving them some illumination you might not expect. Rasterization, REYES, and naïve ray tracing don't provide this sort of effect unless explicitly included, and usually any implementation would suffer from being fairly inaccurate. This is where the beauty of path tracing comes in. It fully simulates this effect naturally through its

algorithm, meaning you get this global illumination effect out of the box. This effect is highly desired in many industries as it provides extremely valuable realism to a computer-generated image. However, a problem in this method is that a computer cannot produce an image if it is busy infinitely bouncing rays off of objects. Methods were invented to accurately simulate this while still being able to produce an image.

## Monte Carlo Method

Monte Carlo methods are broadly defined in mathematics as computation algorithms that rely on recursive stochastic sampling to achieve usable results. This idea was used in tandem with path tracing in order to accurately simulate light bouncing in a scene. The algorithm can simply be defined as follows.

```
method monte_carlo_path_tracing:
    input ray

    trace ray, find point of intersection with nearest surface if one
    exists

    randomly determine whether to stop or scatter the light
    transported by the ray

    if stopping, compute emitted light at this point

    if scattering, randomly scatter the ray according to the type of
    surface the ray has intersected with, and call this method on the
    new ray we've produced
```

Through a series of recursive bounces, the Monte Carlo path tracing algorithm gathers surface data throughout the scene and eventually terminates randomly, successfully simulating global illumination.

## Noise

Unfortunately, Monte Carlo path tracing has its downfalls. Due to its dependency on random termination, the algorithm must be run a large amount of times per pixel in order to produce a clean image. If the algorithm is only run a handful of times, the resulting image will be

riddled with noise and will almost be unrecognizable, and certainly unusable. This is a major issue of Monte Carlo path tracing and the reason why it has taken such a long time for the vast majority of the computer graphics world to adopt it as the primary rendering algorithm. Up until recently, hardware and software limitations meant that it was simply not feasible to use the algorithm in a production. Nowadays, with the most advanced hardware, GPU-accelerated algorithms, and efficient memory methods, Monte Carlo path tracing is being used to render full-length feature films.

Despite the latest technological innovations, using Monte Carlo path tracing to render a fully clean image is an arduous, lengthy process. As recently as 2019, it was reported that a single frame from *Toy Story 4* took up to 30 hours to render [2], and it takes 24 frames to produce 1 second of animation. It is clear to see there is still a lot of time that could be saved in this process.

## **Denoising**

Monte Carlo path tracing can produce a noisy image rather quickly; its bottleneck lies in trying to produce a clear, converged image. Denoising is the process of removing the noise from an image, and if done well, can produce a result that is almost identical to the same image without any noise. This area has been heavily researched so that Monte Carlo path tracing can be utilized without having to deal with the major drawbacks [9]. Many algorithms have been developed to improve the state of denoising for this purpose. Deep learning – a subset of machine learning that has networks capable of learning unsupervised – is a popular framework for achieving efficient and accurate denoising of Monte Carlo path traced images, and is what we'll be focusing on in this thesis.

## *Reconstruction*

Denoising is the process of removing noise from an image, but with extremely low samples, it is increasingly difficult to produce a usable, clear image by only attempting to denoise. Thus, reconstruction is another method that is utilized in order to produce a clear image from a noisy input. Reconstruction involves using iterative algorithms to construct images from the given data. This is also the main focus of Chaitanya et al. [5].

## **VR**

A great deal of research is being done in the area of denoising using deep learning. In particular, research has focused on using a singular image per frame. In virtual reality, two stereoscopic images are being shown every frame – one that pertains to the left eye, and one to the right eye. Both have slightly different viewpoints but are meant to represent a cohesive set of information about the scene you are looking at. The hypothesis of this research project is that by using data from both the left eye and right eye images of an image frame to enhance the performance and results of a deep learning algorithm for denoising Monte Carlo path traced images.

## CHAPTER II

### METHODS

#### **Dataset**

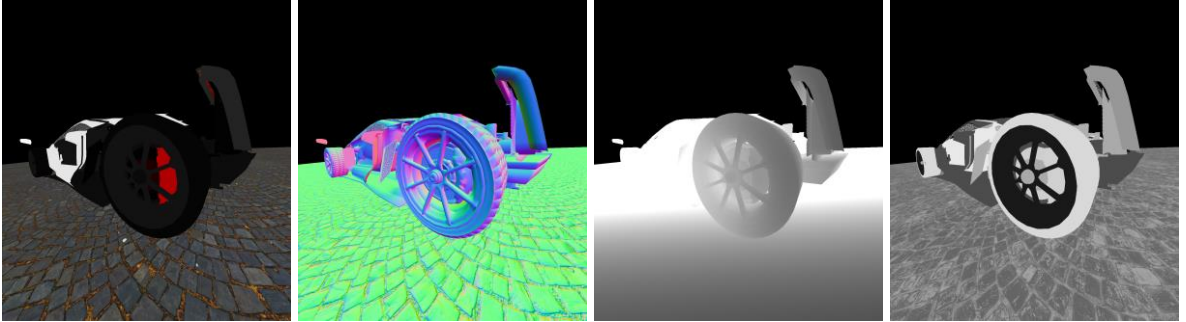
The dataset is imperative for a neural network as it drives how the network learns and what its biases are. For this research it is imperative that the dataset consist of images depicting many different types of scenes in order to train the network in a way that it can handle most inputs. For example, if the dataset didn't contain any scenes with glass, the network would later have trouble interpreting noise related to glass reflections or refractions. There are many other examples of unique materials and effects that come up in rendering, which meant the size of the dataset was important. For each image in the dataset, there had to be several versions with varying degrees of low samples per pixel (spp), and high-quality renders, commonly referred to as ground truth, or GT, images. Because we aim to utilize stereoscopic images, each data element had to consist of an image pertaining to the left eye and another pertaining to the right eye.

#### *Image Features*

On top of having a noisy version of each data element as well as a corresponding GT version, there are other features present in a 3D image that were necessary for our neural network to train with. Each feature is represented as either a vector of float values or a singular scalar float value, and can be viewed visually on their own. They are as follows.

- Albedo – This is the base RGB value of every pixel in the image before any path tracing is done to alter the pixel color based on lighting and shading.

- Normal – This is a set of float values that represent the vector perpendicular to the surface of every pixel in the image. Normally this feature is represented as three values; one each for x, y, and z, but we compressed it to two. [8]
- Depth – This is a scalar value representing the depth of every pixel in the image.
- Roughness – This is a scalar value representing the roughness of the material of every pixel in the image.
- 1spp – This is the RGB values of the path traced image at 1 sample per pixel.
- 2spp – This is the RGB values of the path traced image at 2 samples per pixel.
- 4spp – This is the RGB values of the path traced image at 4 samples per pixel.
- GT – This is the RGB values of the path traced image at 256 samples per pixel, what this research project designated as the ground truth.



**Figure 1.** (From left to right) Albedo, normal, depth, and roughness features shown as individual images.

### *Building the Dataset*

We were provided with a dataset used in another denoising paper by Bako et. al. [3]. The dataset consisted of .json files describing components of the scene featured in each image such as the position and angle of the camera, the texture files of the objects, and other important information. The dataset also contained existing noisy and ground truth images for each .json file, however, for our research, we needed two images per data element. So, we implemented this

mathematical algorithm to properly transform the camera view and position, which can be found below.

```
method get_eye_transform:
    input camera position, look at point, up vector, and constant x

    calculate distance from camera position to the look at point,
    defined as dist

    create vector (x, 0, dist, 1),
    defined as x_trans

    get left eye view matrix by using logic from OpenGL gluLookAt
    function [4]

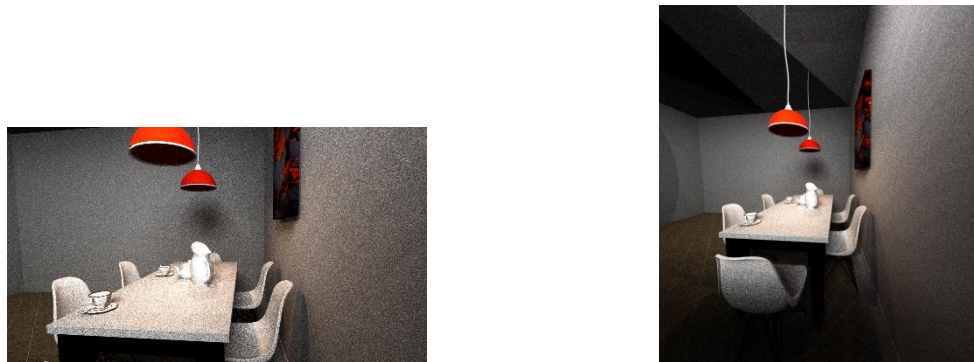
    invert left eye view matrix, defined as C

    matrix multiply C * x_trans as a transformation matrix to get the
    right eye transform vector, defined as re_trans

    matrix multiply dist * C to get right eye look at point,
    redefined as re_lookat

    output re_trans, re_lookat
```

On top of this adjustment, other tweaks were necessary to derive a stereoscopic image from the original data, such as changing the aspect ratio (see example below). We decided to emulate the parameters of an HTC Vive VR headset. Though this seems like a simple process, due to the scale of the dataset, it would have taken an arduous amount of time to manually go in and alter these parameters, which is why a Python script was written to automate this process and allow for straightforward generation and iteration of the dataset elements.



**Figure 2.** (Left): 1920x1080 image with 90° FOV. (Right): 1440x1600 image with 110° FOV (HTC Vive).

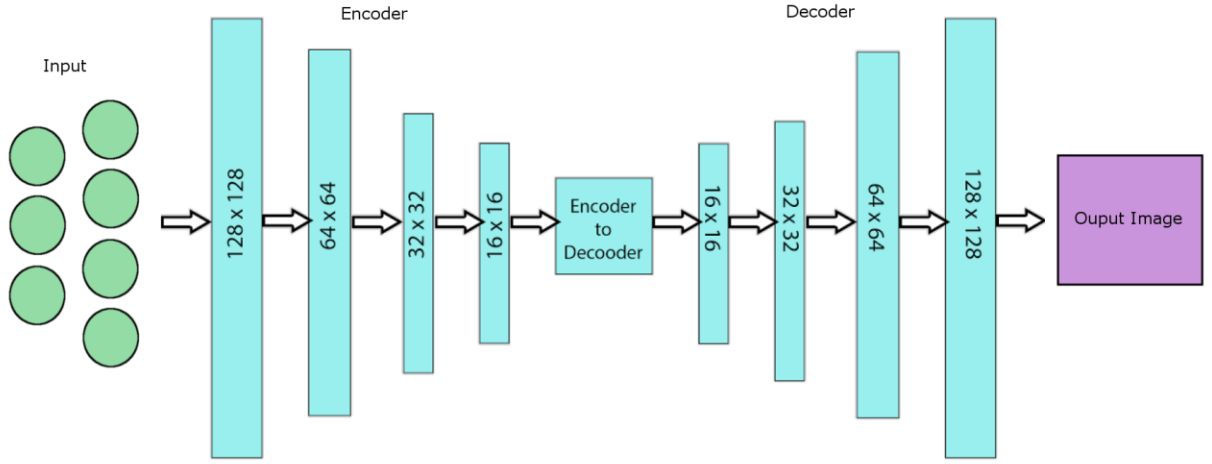
Eventually, after issues with the initial dataset arose from hardware complications, I focused on utilizing a custom dataset using open-source scenes and textures with the 3D software Blender and its subsequent rendering packages, Eevee and Cycles. Using these technologies, the final images for the project was generated.

In order to further enhance the training of the neural network, a tertiary step is taken between generating the images and inputting them into the network. This step involved a preprocessing stage that broke each image into multiple patches, which are essentially just smaller pieces that all together make up the whole image. Though there is technically no new data being introduced through this process, by inputting the images into the network as multiple patches, each patch is considered its own unique image with its own unique set of features, and sometimes there can be several differences in features between different patches from the same image, so the network is able to learn more comprehensively this way. Furthermore, data augmentation can be done to increase the amount of data in another way. This involves simply rotating the patches in different ways and treating them as separate elements in the training set.

### **Network Architecture**

Chaitanya et al. [5] developed a neural network to ‘reconstruct’ an image, utilizing an architecture that included an encoder for the first section of layers, which passed its results into a decoder for the remaining layers, and skip connections between the network layers. This architecture helped them achieve impressive results on short sequences of animation. For the purposes of this research, this network was a good starting point, but there were several major implementation changes we were going to need to make to test the hypothesis of the project.





**Figure 3.** A diagram showcasing the basic architecture of the neural network based on Chaitanya et al. [5].

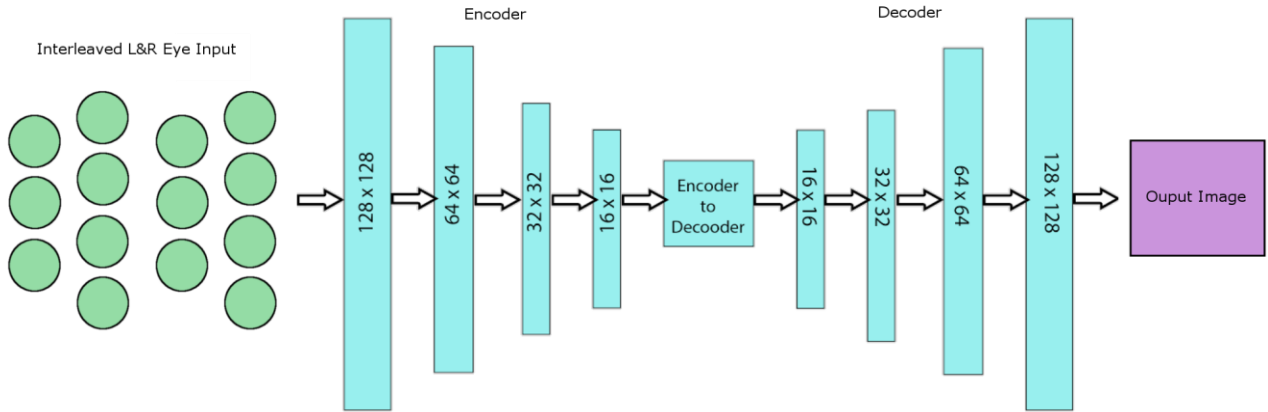
The dilemma mainly lied in the input structure. For the implementation from Chaitanya et al., there were 7 scalar values per pixel being used; the noisy RGB values, view-space shading normals in the form of a 2D vector, depth, and roughness. The network in the paper is structured to fit this type of input. The goal was to develop a method that would allow the network to take in this set of 7 values per pixel for both the left eye and right eye image of a data element, and on top of that, use this additional data to improve the accuracy and efficiency of both the training process and the actual performance of the resulting denoiser. To further test the limits of this method, we sought to use extremely noisy input images; around 1 or 2 spp. The theory behind this was that since two images are being fed into the network rather than one, we can afford to provide noisier input.

### Experiment 1 – Concatenated Features

We decided to expand the input layer of the network architecture to contain both the left eye and right eye feature data, using an interleaving structure as follows.

[normal\_left\_eye, normal\_right\_eye, depth\_left\_eye,  
depth\_right\_eye, roughness\_left\_eye, roughness\_right\_eye, ... ]

After making the necessary adjustments to the original implementation from Chaitanya et al. [5] in order to get these new input structure to work, we decided to test how the network would perform with this as the only major change. In this experiment, the left eye and right eye data is directly infused into the network as a concatenated array of features. The rest of the network architecture remained the same as the original implementation.



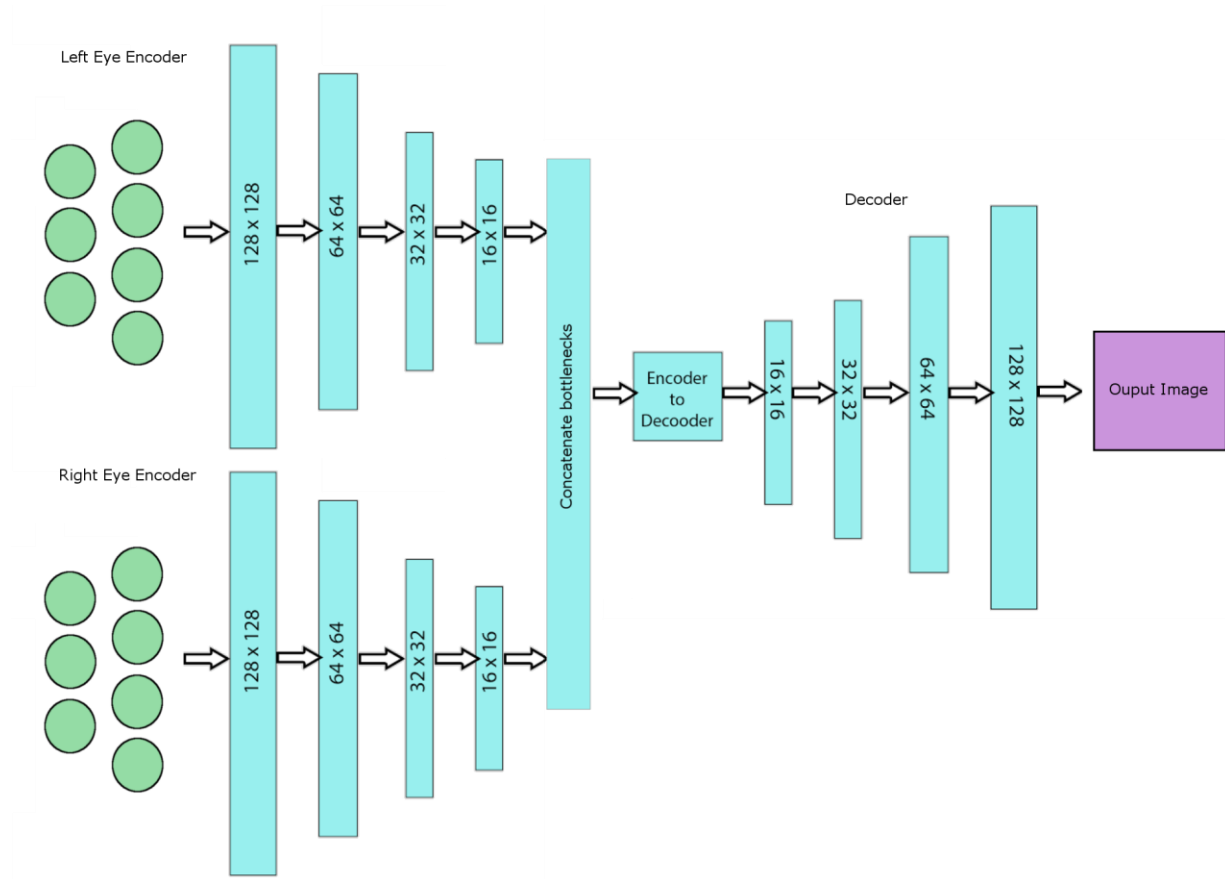
**Figure 4.** A diagram showcasing the architecture of the network for experiment 1.

## Experiment 2 – Separate Encoders

In the original implementation there is one encoder for all the input, and one decoder. The focus of this research project was utilizing data from two images per frame, both left eye and right eye images. Though both images were to be fed into the network, we wanted to try another major alteration to the network that could possibly better facilitate learning using both data elements. This idea involved adding a second encoder to the neural network.

For this experiment, both the left eye and right eye images were still being passed into the network as input, but the network processed each eye separately through a unique encoder,

denoted as the ‘left eye encoder’ and the ‘right eye encoder’. Then, the network takes the outputs from both encoders, concatenates the bottlenecks into one tensor, and passes that into the one decoder that has been present in all iterations of the architecture.



**Figure 5.** A diagram showcasing the architecture of the network for experiment 2.

## Performance Evaluation

The goal of these experiments is to analyze different network architectures and determine which can best utilize stereoscopic image data to improve the quality of denoising. In order to accurately measure the proficiency of these experiments, a way to evaluate performance was necessary. There are two avenues in which performance is normally measured relating to denoising algorithms: numerical and visual evaluation. For visual analysis, the process is as

simple as running the fully trained networks on a set of test images and visually comparing the results with the GT counterparts, and to the results of the other networks.

The numerical evaluation of the results of a network involves evaluating how close each pixel's RGB values are to the GT image's RGB values. There are several methods for accomplishing this task, but for the purposes of this thesis, we used RMSE (root mean square error) and SSIM (structural similarity index). For RMSE, the closer the result value is to 0.0, the better. For SSIM, the opposite applies – a value closer to 1.0 is desired.

### *Constructing the Validation Set*

An ideal validation set is a set of images from scenes with varying features not yet depicted. As was mentioned in the introduction of this thesis, it's imperative for a network to be exposed to different sorts of scenes with varying light interactions, i.e. a scene with glass in it or a scene with motion blur. This way, the network learns how to denoise these irregular features, and doesn't produce undesirable artifacts when given images containing these features. Thus, we sought to choose scenes out of the dataset that were different enough from each other to test the capability of the networks to handle various different types of imagery.

## CHAPTER III

### RESULTS

#### Numerical Analysis

As explained above, we evaluated the different architectures using RMSE and SSIM.

After running each network on a test set of about 20 images, the results were as follows.

**Table 1.** SSIM and RMSE results for each network.

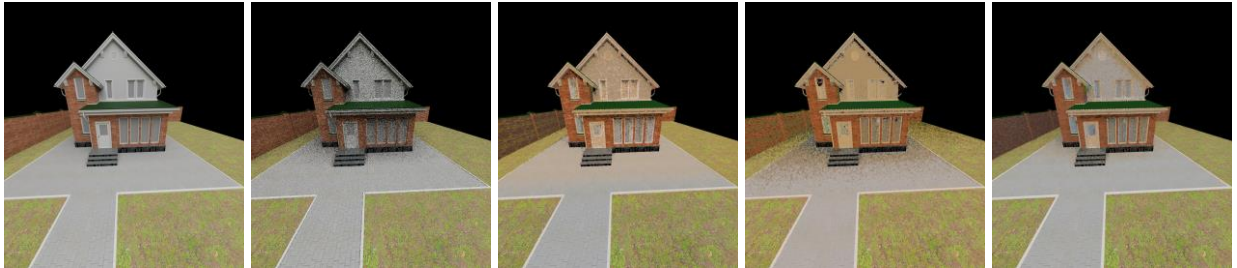
	Base	Experiment 1	Experiment 2
<b>SSIM</b>	0.7298978734201695	0.623899529986208	0.7637802089940835
<b>RMSE</b>	0.1009405611981445	0.131044029002343	0.0862252814807986

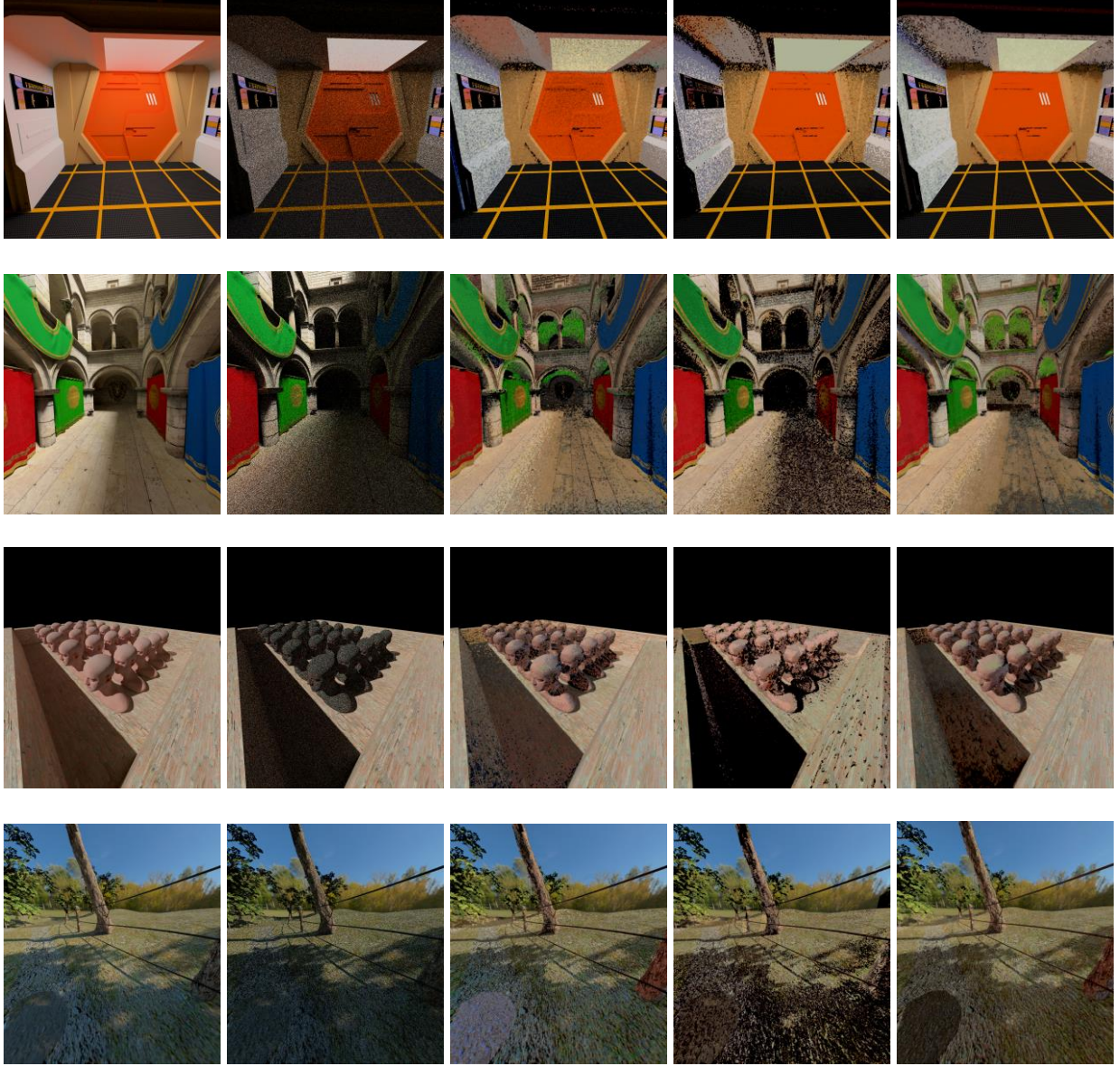
Examining these numerical results, Experiment 2 was the best performing network of the three, with an SSIM of about 0.764 and RMSE of 0.086, beating out the base network.

Experiment 1, although containing the same stereoscopic image data as Experiment 2, failed to perform as well and produced metrics lower than the base network.

#### Visual Analysis

Each scene below has the GT image on the far left, the input noisy image in the middle, and each network's output image on the far right, in order of (Base, Experiment 1, Experiment 2).

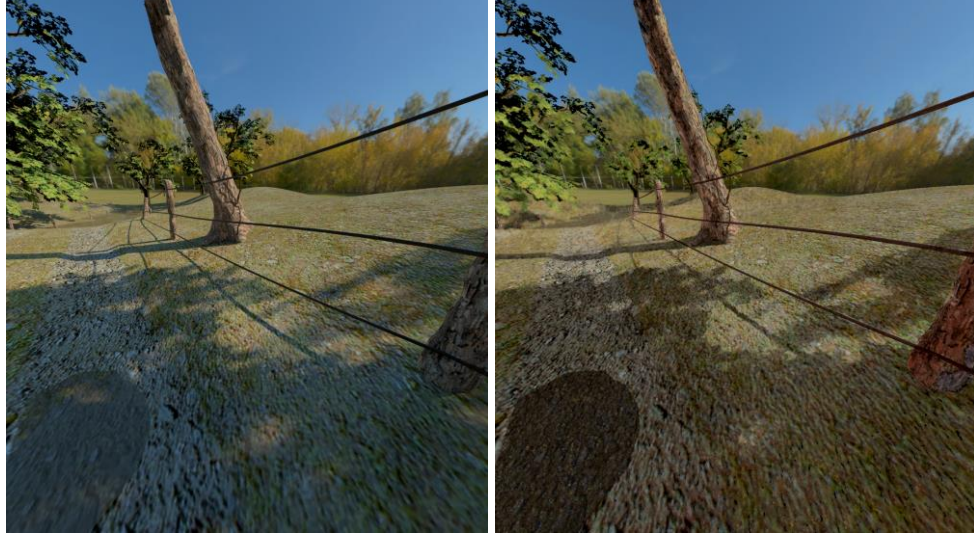




**Figure 6.** (Left to right): GT, Noisy input, Base output, Experiment 1 output, Experiment 2 output.

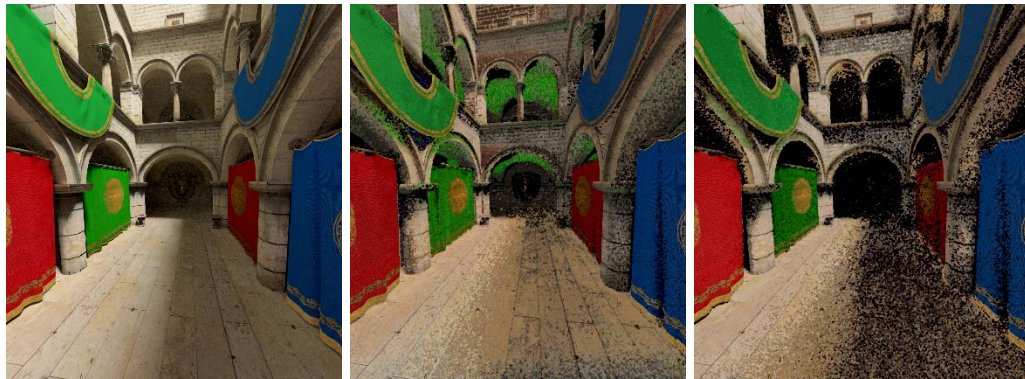
The visual results of these different scenes are varied, and despite the numerical results, each image produced by Experiment 2 does not clearly look the most like the original GT image compared to the other two networks. For the below scene, the differences between the GT image on the left and the result image from Experiment 2 are less than the other two networks, which is expected according to the numerical evaluation.





**Figure 7.** Close-up comparison between GT image (left) and output of Experiment 2 (right).

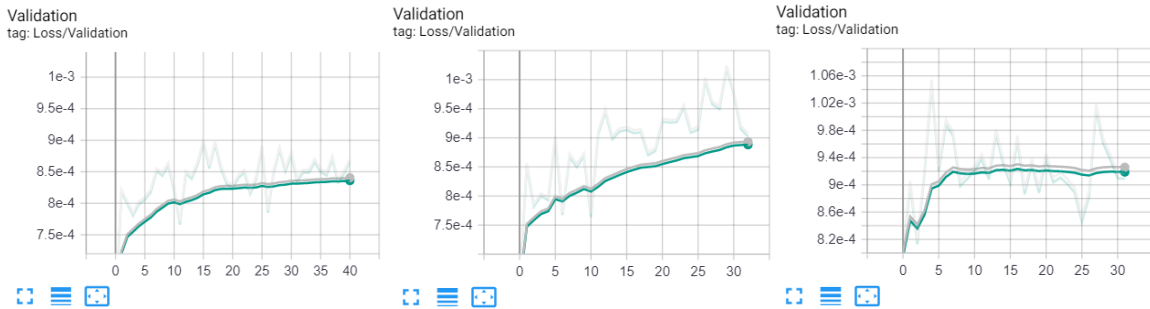
In the scene below, there was more visual variation between the experiments and the base. Although the base network (middle) reproduced the look of the ground and corridors much more closely than Experiment 1, there are large, noticeable green artifacts in the upper corridors and middle lower corridor, which aren't present in Experiment 1's result.



**Figure 8.** Comparison between GT image (left) and output of Base (middle) and Experiment 1 (right).

## Validation Analysis

Each network architecture's validation graph is shown below, in order from left to right of (Base, Experiment 1, Experiment 2). The validation of a neural network involves computing the error on the validation set multiple times throughout the course of training. The closer a network's validation graph slope gets to zero, the closer it is to converging, which is the desired stopping point for training a neural network.



**Figure 9.** (Left to right): Validation graph of base, Experiment 1, and Experiment 2.

These graphs all contain two different plots. The green plot represents the spatial loss, and the gray plot represents the gradient loss. Spatial loss is equivalent to L1 loss and provides a good image metric that is tolerant to outliers [5]. Gradient loss is similar to edge detector loss and is an image comparison metric from the medical imaging domain [5]. The faded versions of the plots represent the plot before increasing the smoothing value – the faded versions are smoothed at a value of 0.6 and the solid versions are smoothed at a value of 0.99.

Each of these graphs show the graph where the validation goes up over time, which is a sign that the network is overfitting, which means the model is too closely fitting to the set of input data. This is commonly caused by a dataset that needs to be increased in size, which is something that is covered in the Conclusion & Future Work section below.



## **CHAPTER IV**

### **CONCLUSION & FUTURE WORK**

#### **Conclusion**

In this thesis, the objective was to explore various methods for enhancing results of deep learning denoising algorithms by modifying the architecture of convolutional neural networks in different ways to best utilize the additional data provided when dealing with stereoscopic images. This objective is important in the world of computer graphics as Monte Carlo path tracing is one of the most popular and highly used methods for rendering 3D imagery today, and as hardware continues to progress, real-time products such as VR experiences and video games will attempt to further integrate path tracing into their rasterization-focused render pipeline. Finding a way to efficiently and accurately denoise path traced images is imperative in order to allow these mediums to use this costly algorithm to simulate real-world lighting effects and interactions while maintaining a strict frame rate.

Based on the results from the two experiments performed on the network architecture, there looks to be potential in certain architecture modifications for optimally utilizing stereoscopic image data. Experiment 2 produced numerical results that improved upon the base implementation, and in most cases, the output image of Experiment 2 was comparable or better to the base implementation. In the context of a VR application, this stereoscopic data will naturally be readily available and based on the explorations of this thesis, it is worthwhile to use this data within the deep learning denoising pipeline in order to enhance the final image.

## **Future Work**

There are many areas of improvement that I believe could enhance the results of this research project and would be worth pursuing in future works.

### *Data Size*

An oft-reported method of improving performance of deep neural networks is adding more data to your training set. It is time-consuming to generate frame data from scenes with a wide array of different features and viewpoints. In the future, generating more data to increase the size of the data set could provide a boost to performance.

### *Tweaking Hyperparameters*

Another way to enhance performance of deep neural networks is tweaking your hyperparameters – this includes values such as the learning rate, batch size, etc. It is difficult to determine the best hyperparameters for a neural network without trial and error, which is a process that takes time. In the future, testing these networks on different hyperparameters may lead to improved results.

### *GT Quality*

The quality of the GT images essentially determines the level of quality the network will be trying to learn. In this thesis, our dataset consisted of GT images rendered at 256 samples per pixel. Usually, final frames of feature films and other productions which use Monte Carlo path tracing generally use sample counts of up to tens of thousands. From the Introduction section of this thesis it's known that generating images at such a high sample rate is an enormously time-consuming task, which was the thought process behind using 256 spp images for the GT in this thesis. However, using a GT image with noticeable noise makes it more difficult for a neural network to learn the right relationships and can result in undesirable artifacts in the final image

that potentially cannot be fixed by other methods described in this section. For future work, higher quality GT images would likely improve the results.

#### *Super-sampled Albedo*

For the dataset of this thesis, the albedo pass was rendered using Blender’s GPU renderer ‘Eevee’. This may have caused minor artifacts along the edges along geometry and within textures in the processed images being used during training of the neural networks. Though they are small and hard to notice without looking for them specifically or zooming in closely, these artifacts can cause similar detrimental effects to what was explained in the above subsection. Super-sampling the albedo pass involves rendering using a path traced rendered such as Blender’s Cycles and could be a potential fix to this problem.

## REFERENCES

- [1] P. Christensen, J. Fong, J. Shade, W. Wooten, B. Schubert, A. Kensler, S. Friedman, C. Kilpatrick, C. Ramshaw, M. Bannister, B. Rayner, J. Brouillat, and M. Liani, “RenderMan: An Advanced Path Tracing Architecture for Movie Rendering”, *ACM Transactions on Graphics (TOG)* 37, 3, Article 30 (July 2018)
- [2] Kyle Desiderio and Ian Phillips, “How Pixar’s animation has evolved over 24 years, from ‘Toy Story’ to ‘Toy Story 4’”, *Insider*, [Online] Available: <https://www.insider.com/pixars-animation-evolved-toy-story-2019-6> [Accessed: 2-Mar-2020]
- [3] S. Bako, T. Vogels, B. McWilliams, M. Meyer, J. Novak, A. Harvill, P. Sen, T. DeRose, F. Rousselle, “Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings”, *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2017)*
- [4] OpenGL gluLookAt [Online] Available: <https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/gluLookAt.xml> [Accessed: 27-Jan-2020]
- [5] C. R. Alla Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, T. Aila, “Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder”, *ACM Transactions on Graphics (TOG)* 36, 4, Article 98 (July 2017)
- [6] Kajiya, James T. "The rendering equation." *Proceedings of the 13th annual conference on Computer Graphics and Interactive Techniques*. 1986.
- [7] Pharr, Matt, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.
- [8] Valient, Michael, “Deferred Rendering in Killzone 2”, *Develop Conference*, July 2007.
- [9] Zwicker, Matthias, et al. "Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering." *Computer Graphics Forum*. Vol. 34. No. 2. 2015.